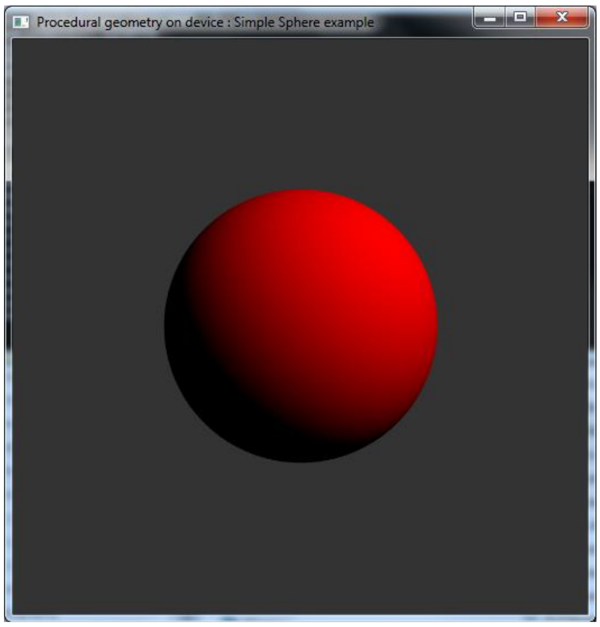
GigaSpace Tutorials

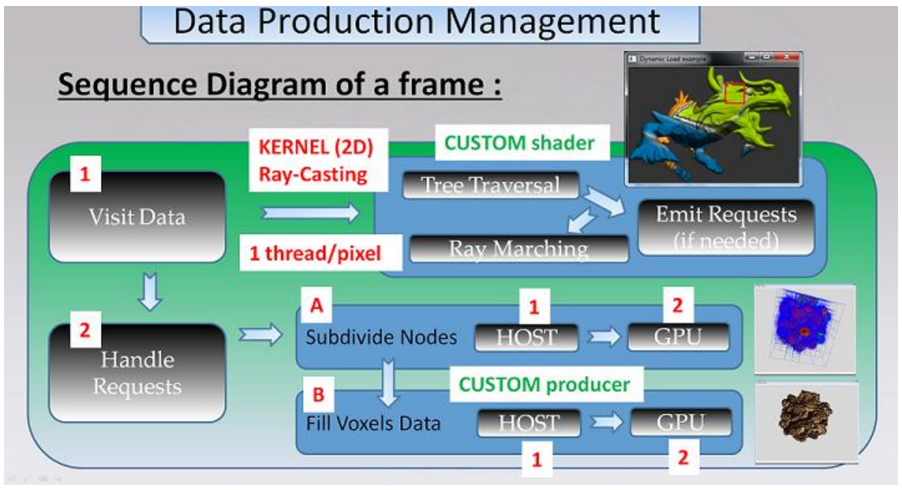
1. Procedural Generation on Device (GPU)

A. Simple Sphere tutorial

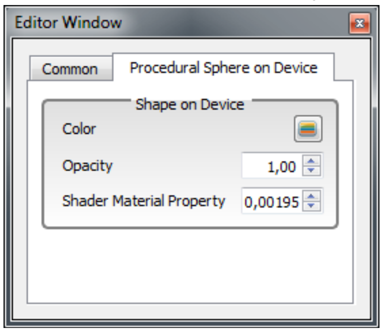
This example shows the usage of a GPU producer used to generate a simple sphere.



In the GigaSpace sequence pipeline, “color” and “normal” are generated by the GPU Producer on-the-fly :



The Renderer is a classical ray-marcher, and is done brick by brick.


Editor Window

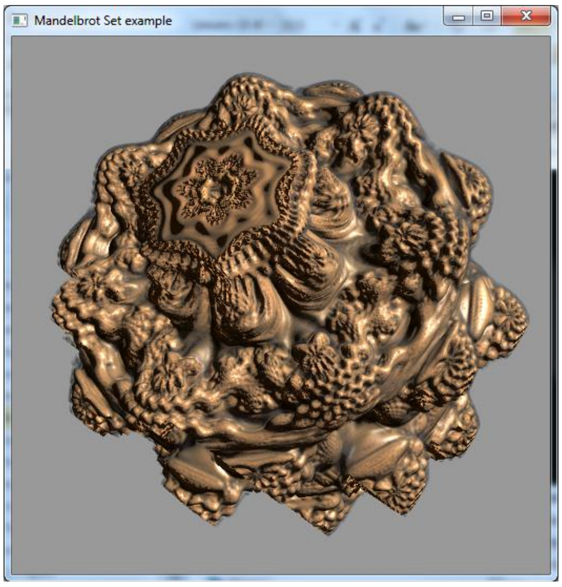
Common | Procedural Sphere on Device

Shape on Device

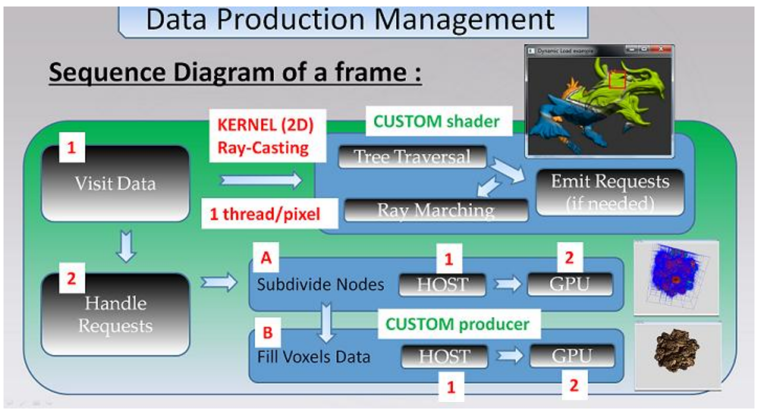
Color	
Opacity	1,00
Shader Material Property	0,00195

B. Complex Scene tutorial

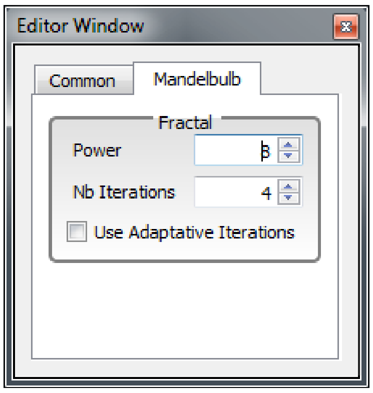
Based on the same model as the simple sphere, this example shows the usage of a GPU producer used to generate procedurally a fractal object, a mandelbulb.



In the GigaSpace sequence pipeline, “color” are generated by the GPU Producer on-the-fly. Shading compute normal by central finite differences at brick-marching.



The editor window enables you to modify the scene :


Editor Window

Common | Mandelbulb

Fractal

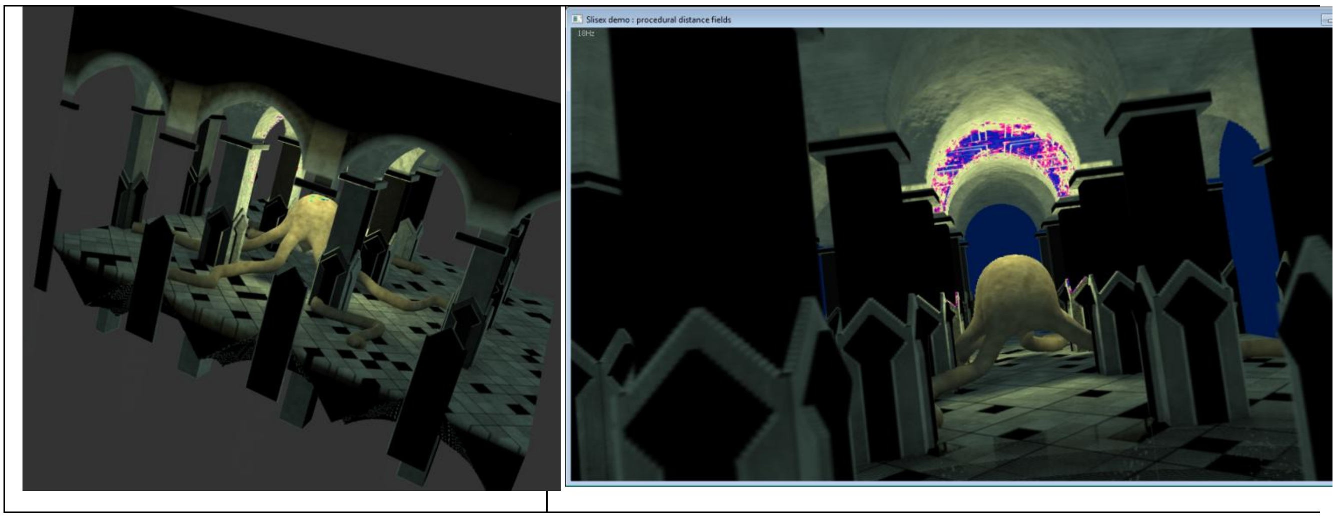
Power 8

Nb Iterations 4

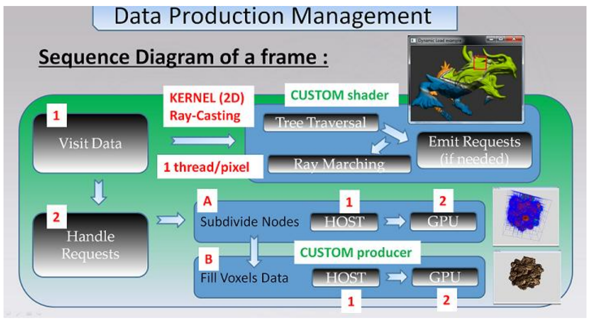
☐ Use Adaptative Iterations

C. Complex Scene tutorial

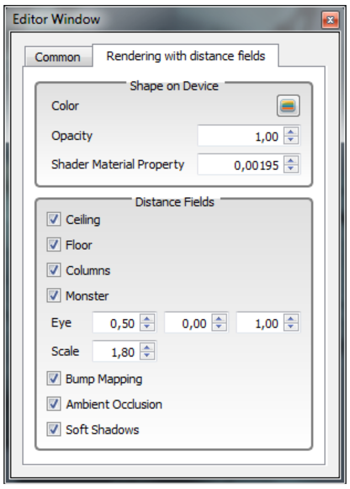
Based on the same model as the simple sphere, this example shows the usage of a GPU producer used to generate procedurally a complex scene. It is based on a demo by Inigo Quilez using Distance Field technics.



In the GigaSpace sequence pipeline, “color” are generated by the GPU Producer on-the-fly. Shading is also done at Production time (shadow, noise, etc…)



The editor window enables you to modify the scene :


Editor Window

Common | Rendering with distance fields

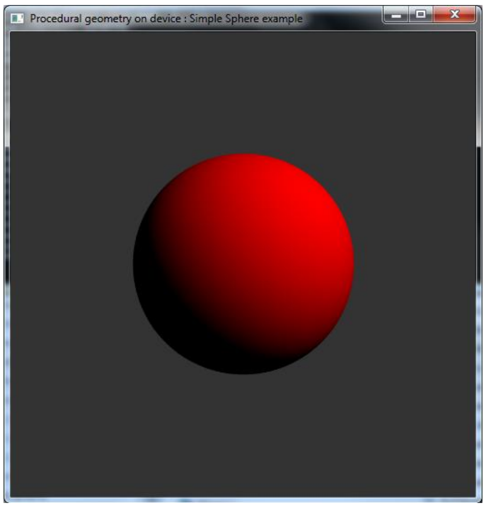
Shape on Device
- Color
- Opacity 1,00
- Shader Material Property 0,00195

Distance Fields
- [✓] Ceiling
- [✓] Floor
- [✓] Columns
- [✓] Monster
- Eye 0,50 0,00 1,00
- Scale 1,80
- [✓] Bump Mapping
- [✓] Ambient Occlusion
- [✓] Soft Shadows

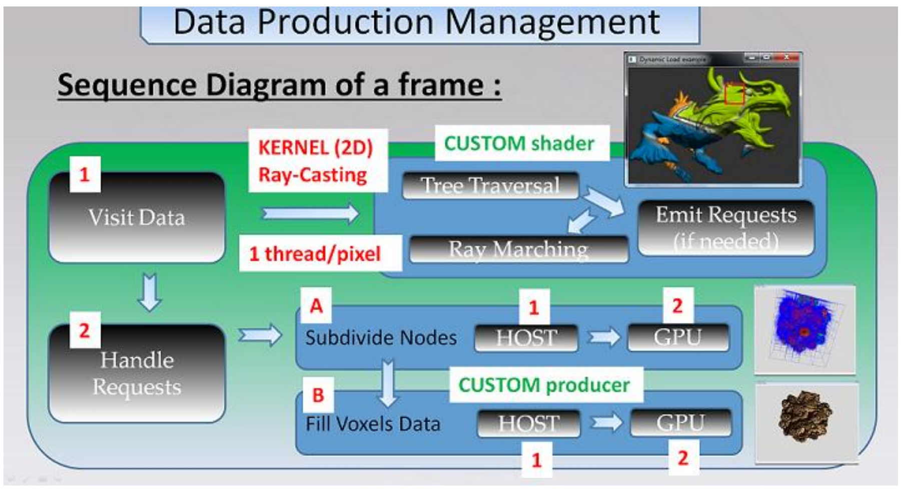
2. Data Streaming from Host (CPU)

A. Simple Sphere CPU tutorial

This example shows the usage of a CPU producer to generate a simple sphere. Data are then streamed to device (GPU) by an associated GPU Producer. To speed / enhance transfer, Memory Mapped is used to share a buffer between CPU and GPU.



In the GigaSpace sequence pipeline, "color" (and eventually "normal) are generated by the CPU Producer on-the-fly. Data are written to a buffer shared by the GPU (memory mapped). The GPU Producer is responsible to read data are write them in GigaSpace cache.

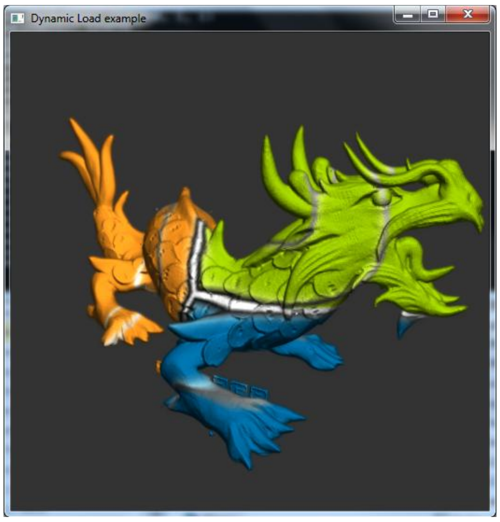


To speed transfer, user has the option to load all files data in a CPU buffer. This enables to use memcpy() instead of using stream IO API like fopen(), fread(), etc...

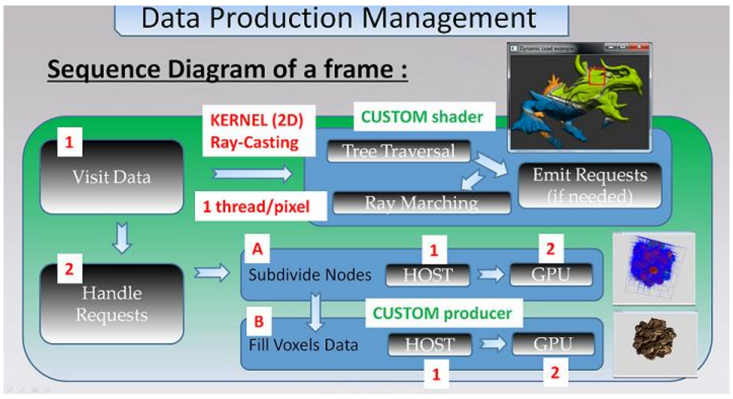
The Renderer is a classical ray-marcher, and is done brick by brick. If normals were not stored in 3D model files, a finite difference is used during brick-marching on-the-fly.

B. 3D Model data streaming tutorial

Based on the same model as the simple sphere , this example shows the usage of a CPU producer to load a pre-voxelized 3D model on GPU. Data are streamed to device (GPU) by an associated GPU Producer. To speed / enhance transfer, Memory Mapped is used to share a buffer between CPU and GPU.



In the GigaSpace sequence pipeline, “color” and “normal” are generated by the CPU Producer on-the-fly. Data are written to a buffer shared by the GPU (memory mapped). The GPU Producer is responsible to read data are write them in GigaSpace cache.

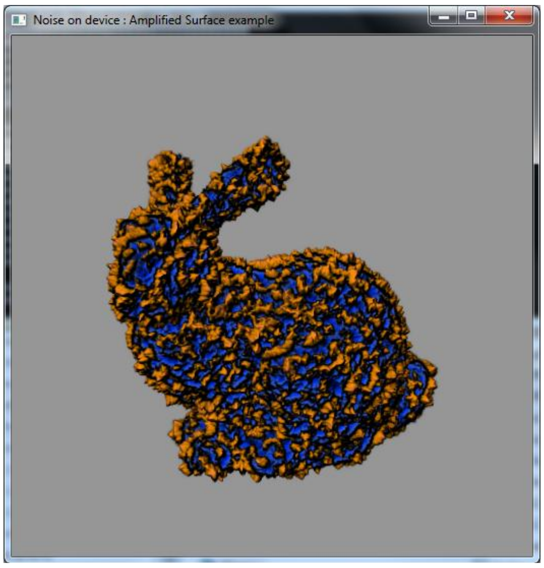


The Renderer is a classical ray-marcher, and is done brick by brick.

3. Data Streaming from Host (CPU)

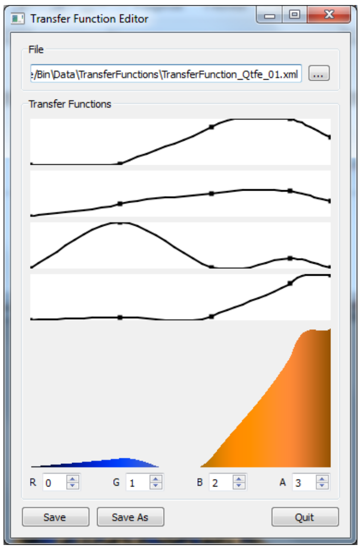
A. Amplified Surface / Volume tutorial

This example shows the usage of a GPU producer to modify 3D surface of 3D model. Data are then streamed to device (GPU) by an associated GPU Producer. To speed / enhance transfer, Memory Mapped is used to share a buffer between CPU and GPU.



The GPU producer has access to a pre-computed "signed distance field" version of 3D mesh of a rabbit (at each 3D points, we have the distance to surface + the normals stored in a float4 read-only texture). Noise is Perlin noise.

User can modify noise appearance by means of a LUT. In this exemple, LUT is applied at Production time. Cache is cleared at each modification

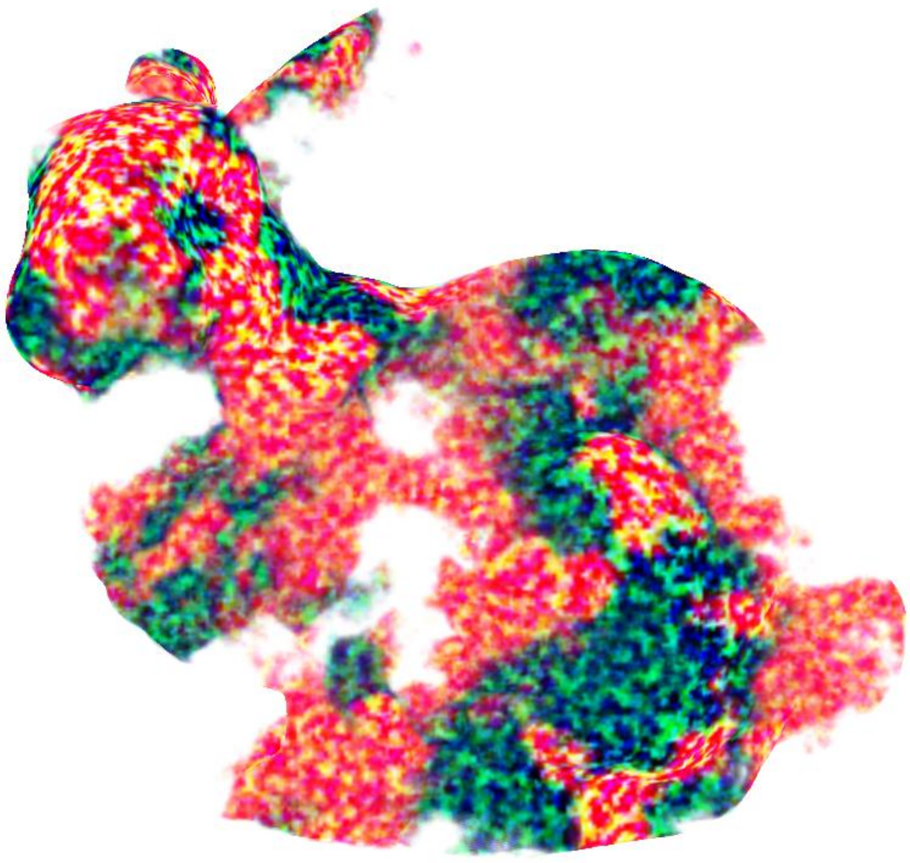


"Amplified Surface" modifies surface, "Amplified Volume" add 3D volumetric effects using alpha.

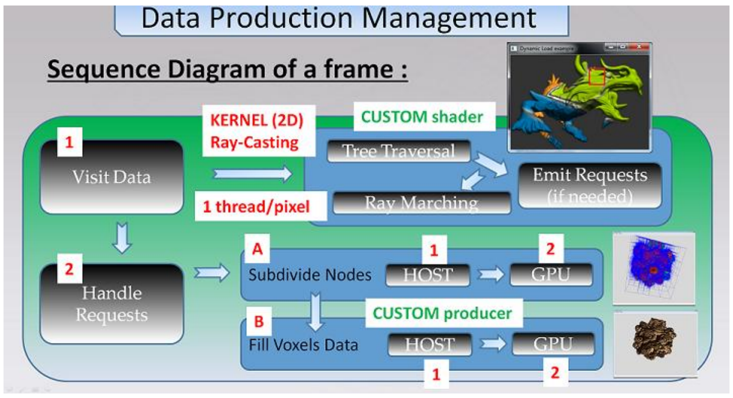
4. Proxy Geometry

A. Noise in a Shell

This demo is based on the same tutorial as the “Amplified Surface” (a pre-processed 3D mesh is stored as Signed Distance Field : distance + normals at each 3D points in a read-only texture). But this time we use a customized Ray-Caster that replace the classical Intersection Box test by retrieving the Shell of a rasterized 3D model.
OpenGL is used to rasterized front and back faces in order to obtain a shell representation of the 3D mesh. Rays start from this shell, not the classical GigaSpace Bounding Box.



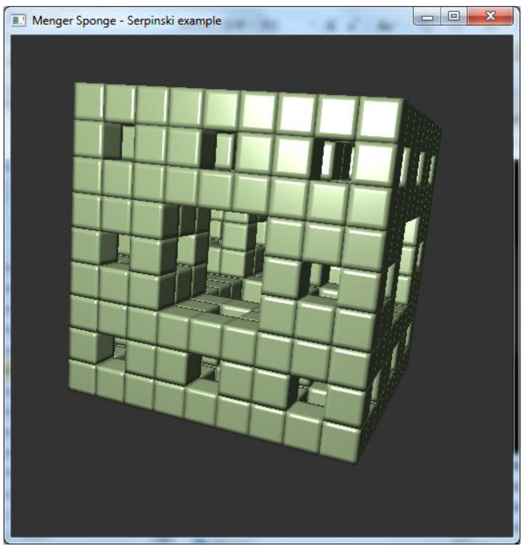
In this demo, Perlin noise is added a GPU Producer stage. Rendering stage does classical ray-marching.



5. Other Data Structures

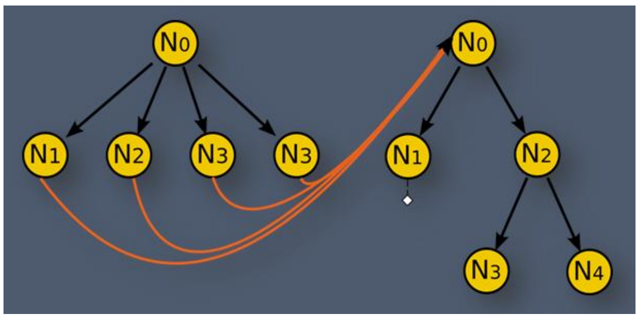
A. Menger Sponge tutorial : 3x3x3 voxels per node

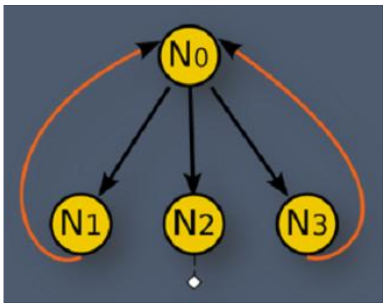
Based on the same model as the simple sphere, this example shows the usage of a GPU producer used to generate procedurally a fractal object, a menger sponge. This mathematical object is not an octree, but a 3x3x3 voxels structure with holes inside.



Here, only 1 brick of voxels is produced but the node data structure loop infinitely on itself : nodes are subdivided but points on the same brick.

Its common in node based data structure have loops to create complex scenes / objects :

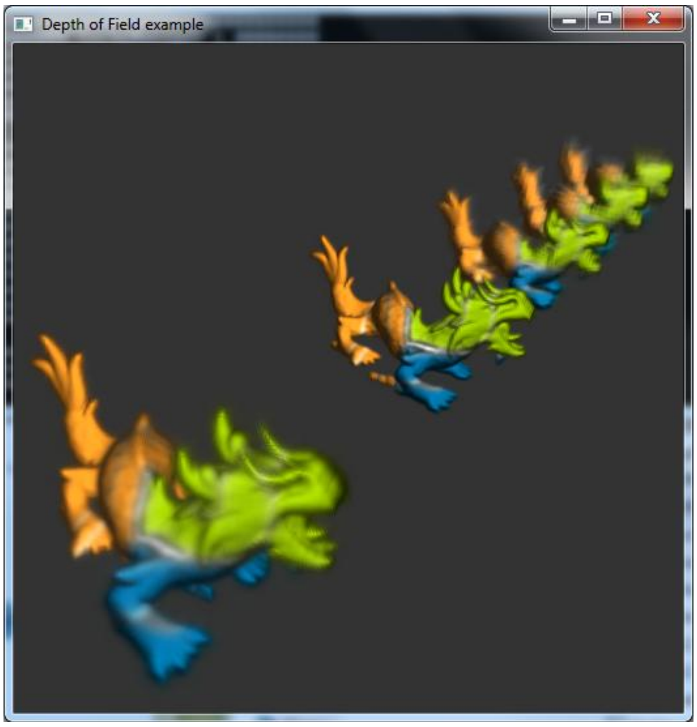




6. Optical Effects

B. Depth of Fields

Based on the same tutorial as the “dynamlic load” (data streaming of 3D pre-voxelized 3D model), this example shows the usage of a GPU Shader used to mimics a depth of field effect

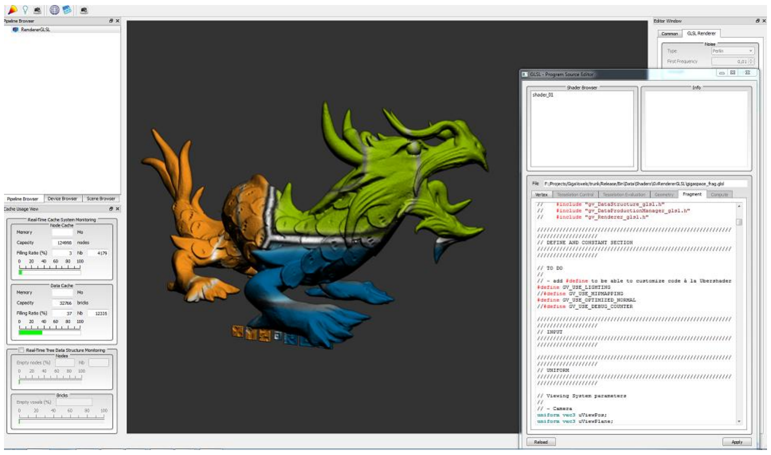


The goal here is to reproduce the “circle of confusion” mechanism.
The shader is responsible to modify the cone-tracing behavior. User customizable API provides access to a cone aperture function modeling lens effect.

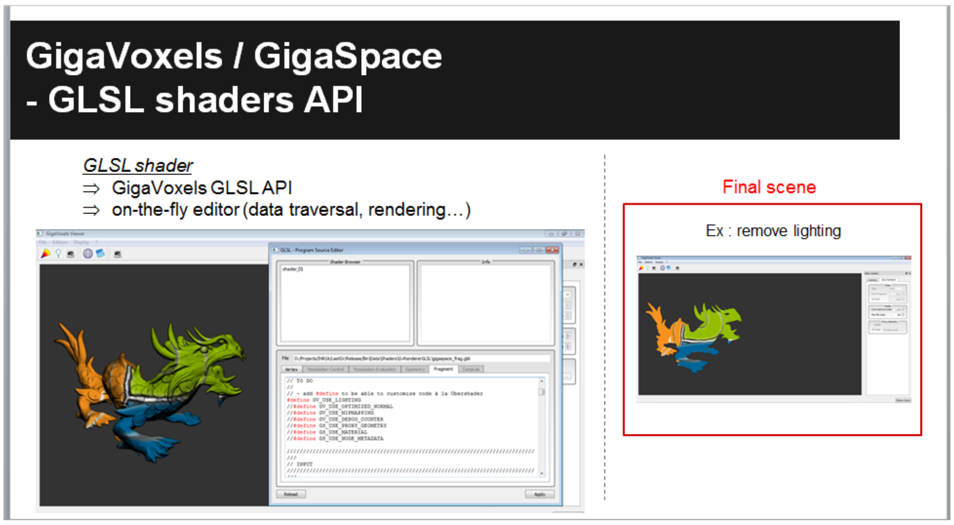
7. Graphics Interoperability

A. GLSL Renderer : data streaming (3D model)

Based on the basic Dynamic Load tutorial to stream data from CPU (a pre-voxelized 3D model), this demo shows how to use the OpenGL GigaSpace renderer and its associated GLSL API to be able to customize on-the-fly the Ray-Casting stage (ray init, intersection, data structure traversal, brick visiting, etc...)


GLSL - Program Source Editor
Shader Browser
Info
shader_01
File F:/Projects/GigaVoxels/trunk/Release/Bin/Data/Shaders/GvRendererGLSL/gigaspace_frag.glsl
Vertex
Tesselation Control
Tesselation Evaluation
Geometry
Fragment
Compute
Reload
Apply

```
//    #include "gv_DataStructure_glsl.h"
//    #include "gv_DataProductionManager_glsl.h"
//    #include "gv_Renderer_glsl.h"

////////////////////////////////////////////////////////////////////////////////
////////////////////
// DEFINE AND CONSTANT SECTION
////////////////////////////////////////////////////////////////////////////////
////////////////////

// TO DO
//
// - add #define to be able to customize code à la Ubershader
#define GV_USE_LIGHTING
//#define GV_USE_MIPMAPPING
#define GV_USE_OPTIMIZED_NORMAL
//#define GV_USE_DEBUG_COUNTER

////////////////////////////////////////////////////////////////////////////////
////////////////////
// INPUT
////////////////////////////////////////////////////////////////////////////////
////////////////////

////////////////////////////////////////////////////////////////////////////////
////////////////////
// UNIFORM
////////////////////////////////////////////////////////////////////////////////
////////////////////

// Viewing System parameters
//
// - Camera
uniform vec3 uViewPos;
uniform vec3 uViewPlane;
```



GigaVoxels / GigaSpace
- GLSL shaders API

GLSL shader
⇒ GigaVoxels GLSL API
⇒ on-the-fly editor (data traversal, rendering…)

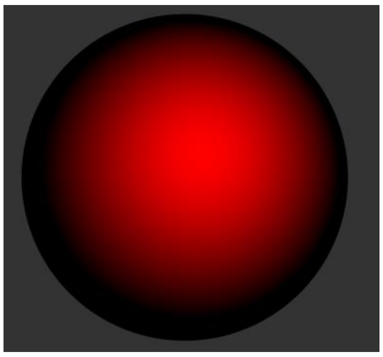
Final scene

Ex : remove lighting

```
// TO DO
//
// - add #define to be able to customize code à la Ubershader
#define GV_USE_LIGHTING
//#define GV_USE_OPTIMIZED_NORMAL
//#define GV_USE_MIPMAPPING
//#define GV_USE_DEBUG_COUNTER
//#define GS_USE_PROXY_GEOMETRY
//#define GS_USE_MATERIAL
//#define GS_USE_NODE_METADATA

////////////////////////////////////////////////////////////////////////////////
///
// INPUT
////////////////////////////////////////////////////////////////////////////////
```

B. GLSL Renderer : procedural shape on device (GPU)

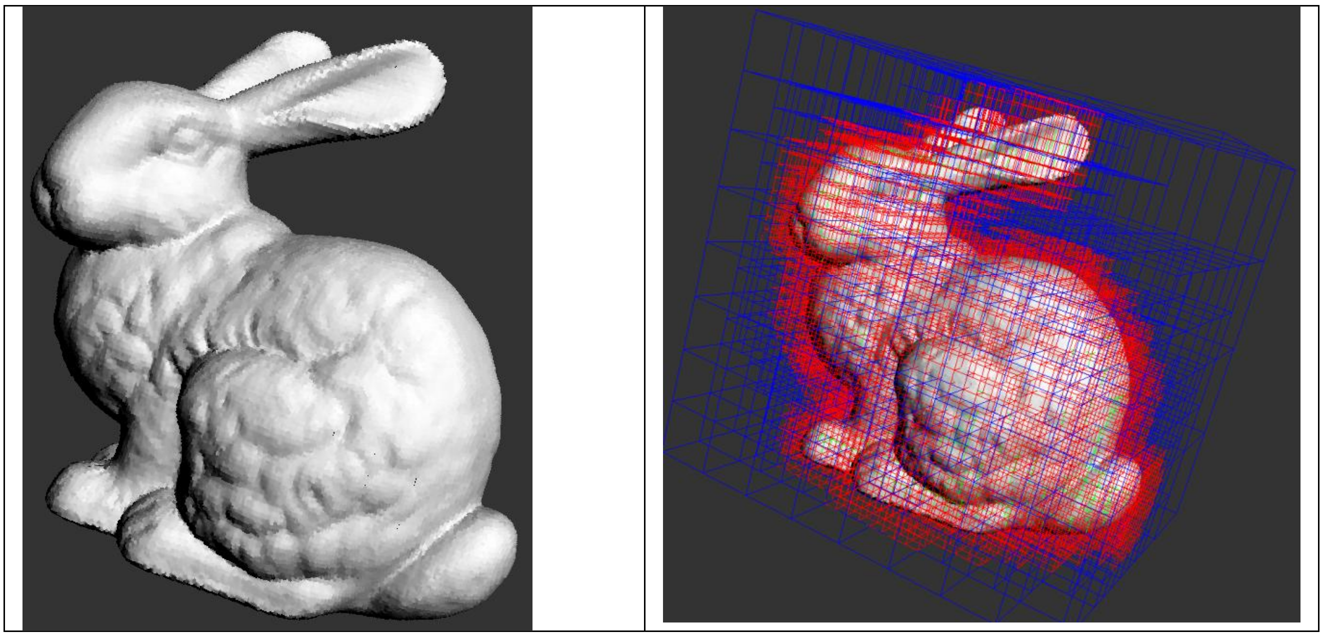
Same idea than the previous tutorial (GLSL Ray-Caster), but based on the basic Simple Sphere tutorial to produce data on-the-fly with the help of a GPU Producer.



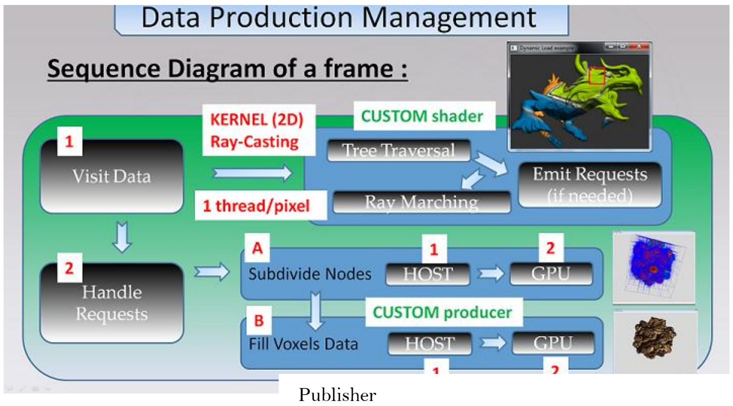
8. Voxelization

A. Basic Voxelization

In this demo, a 3D mesh is voxelized. At each pass (frame), ray-casting stage ask for requests of producing content of empty 3D regions of space. To fill each node, we voxelize the mesh node by node.



This demo is based on the basic “simple sphere CPU” data streaming tutorial where a CPU Producer is used to fill a buffer shared with GPU memory. The GPU Producer read data and fill the associated GigaSpace data cache.

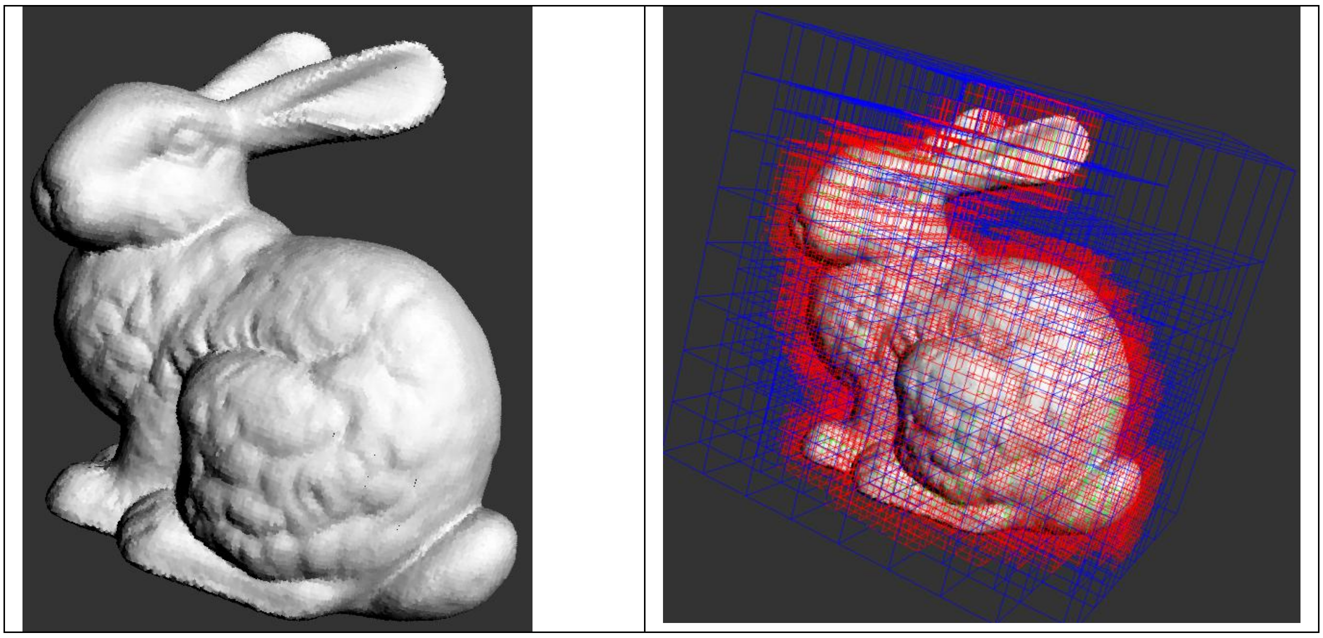


Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr
ISSN 0249-6399

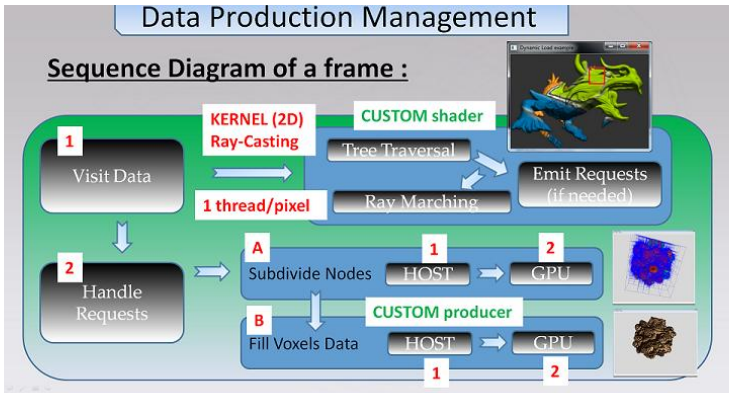
To fill the memory shared buffer during at CPU Producer stage, this time we choose to rely on GLSL shaders to voxelize the mesh. The GigaVoxels API enables to use the CUDA Graphics Interoperability to be able to use its elements in OpenGL/GLSL.
We use brick by brick 3D rasterization by storing normals data with the use of GLSL `imageAtomicAdd()`.
At GPU Producer stage, data is read and place in GigaSpace cache. Then, at shader rendering stage, normal is sampled and normalize and a default color is applied. It' s kind of binary voxelization.

B. Signed Distance Field Voxelization

In this demo, a 3D mesh is voxelized. At each pass (frame), ray-casting stage ask for requests of producing content of empty 3D regions of space. To fill each node, we voxelize the mesh node by node.



This demo is based on the basic "simple sphere CPU" data streaming tutorial where a CPU Producer is used to fill a buffer shared with GPU memory. The GPU Producer read data and fill the associated GigaSpace data cache.

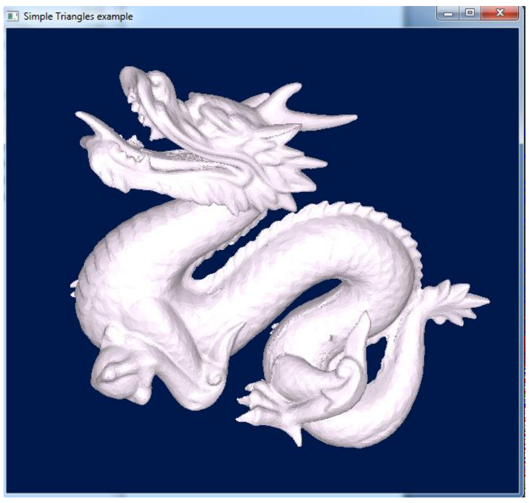


To fill the memory shared buffer during at CPU Producer stage, this time we choose to rely on GLSL shaders to voxelize the mesh. The GigaVoxels API enables to use the CUDA Graphics Interoperability to be able to use its elements in OpenGL/GLSL.
We use brick by brick 3D rasterization by storing a “signed distance field” representation of the mesh.
We store the final closest distance to the mesh at a given voxel.
Then, at GPU Producer stage, data is read and “normals” are computed by central finite differences and stored in GigaSpace cache. Then, at shader rendering stage, normal is sampled and normalize and a default color is applied.

9. GigaSpace

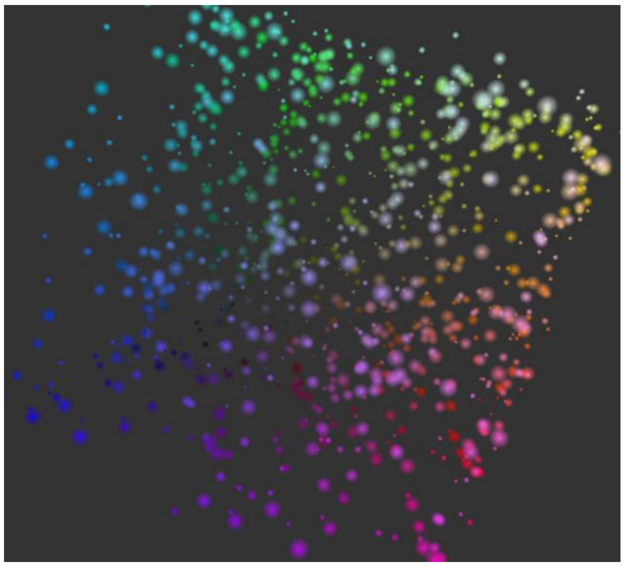
A. Simple Triangles tutorial : managing BVH

Based on the same model as the simple sphere CPU, this example shows how to handle BVH. This demo is more a proof of concept.
Mesh is loaded and pre-process to store a list of triangles in a BVH way. This list of triangles is then traversed at rendering stage.



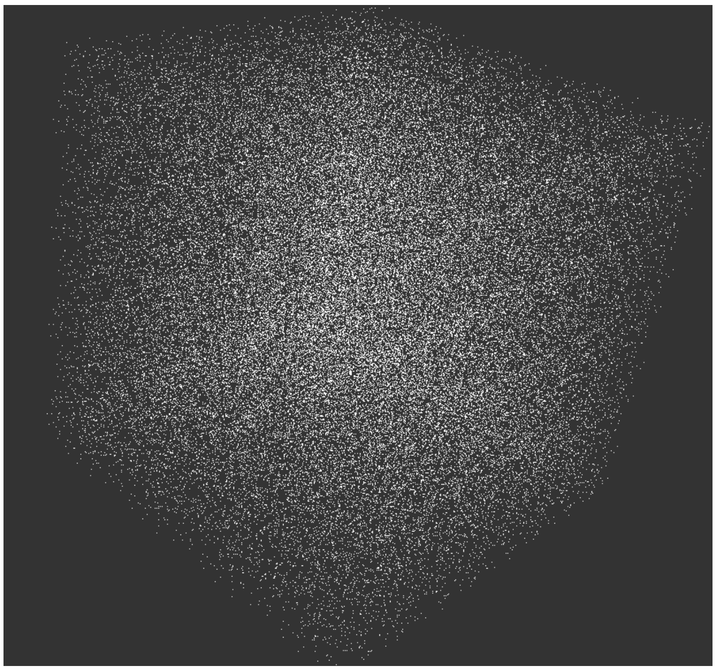
B. Sphere Ray-Tracing : managing points

Based on the same model as the simple sphere CPU, this example shows how to handle BVH. This demo is more a proof of concept.
Mesh is loaded and pre-process to store a list of triangles in a BVH way. This list of triangles is then traversed at rendering stage.



C. Dynamic Frustum Culling : managing VBO

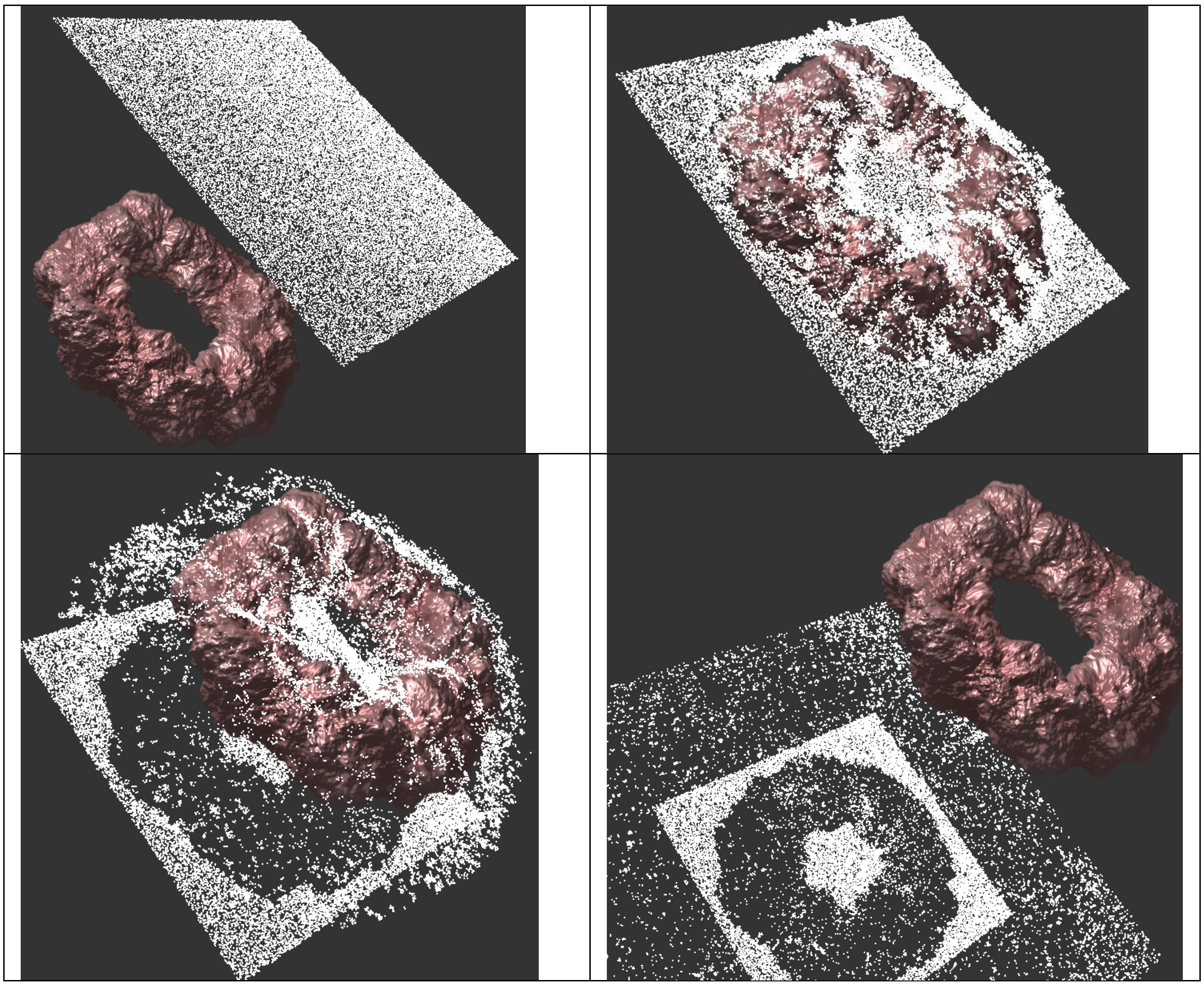
Based on the same model as the simple sphere CPU, this example shows how to handle BVH. This demo is more a proof of concept.
Mesh is loaded and pre-process to store a list of triangles in a BVH way. This list of triangles is then traversed at rendering stage.



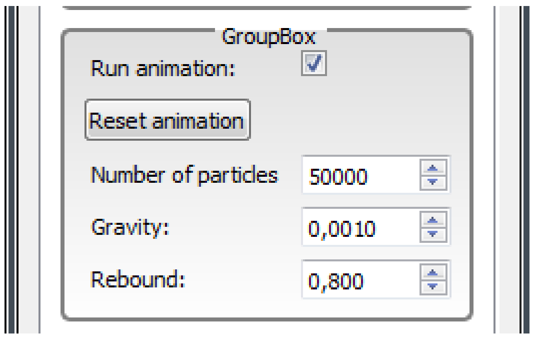
10. Querying the Data Structure

A. Collision Detection

A OpenGL VBO of particles is launched against a GigaSpace object (noise-based torus). The idea is to query the data structure to know if there is data in a region of space, then be able to take a decision (here, collide with particles).



The demo is based on the same tutorial as the simple sphere on GPU, with the add of Perlin noise (a fully procedural GPU Producer). At each frame , a CUDA is launched to update VBO particles by mapping them to CUDA memory (CUDA Graphics Interoperability) and querying the GigaSpace Data Structure.



GroupBox

Run animation: ☑

Reset animation

Number of particles 50000

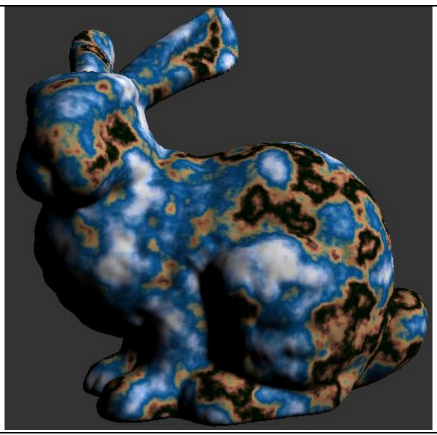
Gravity: 0,0010

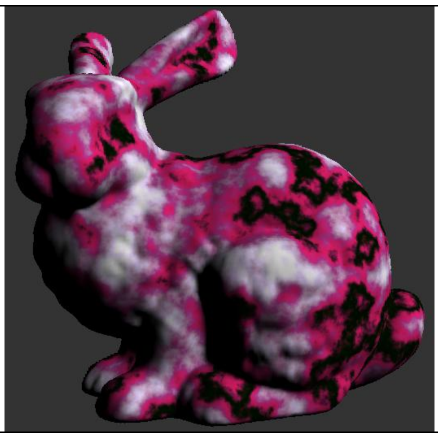
Rebound: 0,800

11. Animation

A. Dynamic Hypertexture

This demo is based on the Amplified Surface demo where a GPU Producer was perturbating a 3D model surface with Perlin noise (model is represented by a Sigend Distance Field : distance to surface + normals at each points). But this time, Noise is applied at Rendering stage where shader apply animation with LUT to access texels.

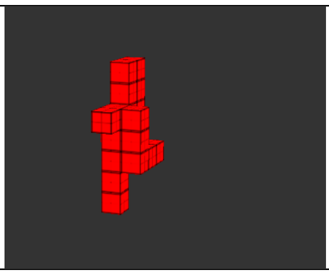


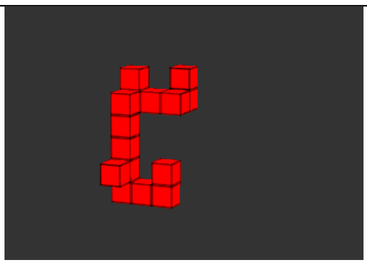


B. Animated Snake

BEWARE : not sure of Jeremy work => should have been like old-style Amiga games with LUT, but here ?

The demo is based on a simple Procedural GPU Producer. Rendering creates animation on-the-fly.

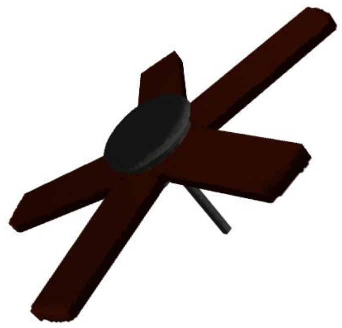


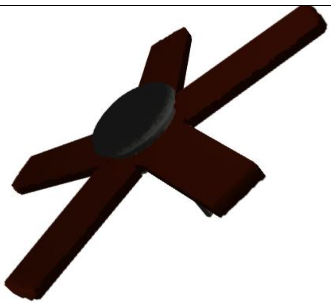


C. Animated Fan

BEWARE : not sure of Jeremy work => should have been like old-style Amiga games with LUT, but here ?

The demo is based on a Data Streamer Producer like the Dynamic Load (loading files from CPU from a pre-voxelized model of Fan). Rendering creates animation on-the-fly.

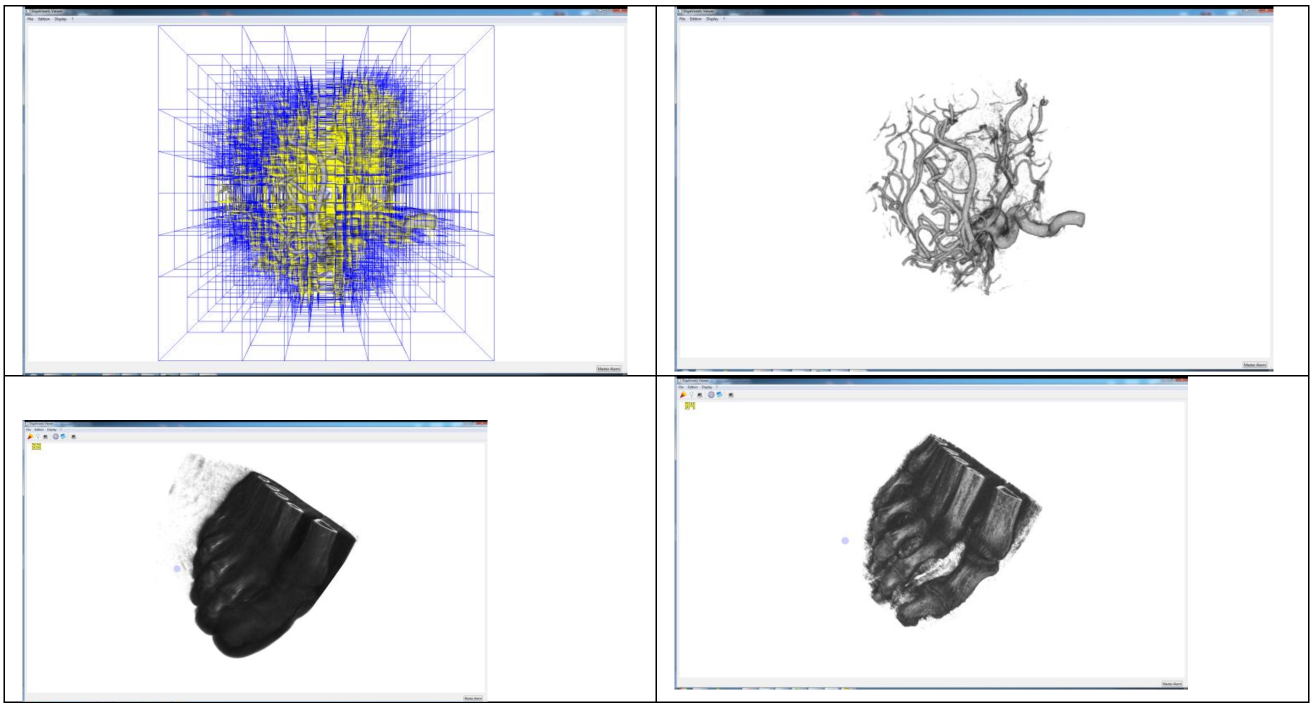




12. Scientific Visualization

A. RAW Data

The demo is based on a Data Streamer Producer like the Dynamic Load (loading files from CPU from a pre-voxelized model of Fan). This demo shows how to customizable the voxelization tool to create voxelized representation of scientific data stored in a basic RAW file. Example : scanner dataset of scalar data.



LUT and threshold can be used to remove noise at production and/or rendering time. Thresholding at production time enables to better fit the octree and accelerate the data structure traversal at rendering stage.

B. DICOM Data

Same as before, for using a dedicated pre-voxelizer tool to handle DICOM data files

13. Instancing

A. Basic Instancing

Blab la bla…

14. Multi-SVO

A. Basic Instancing

Blab la bla…

15. Priorities

A. Basic Instancing

Blab la bla…